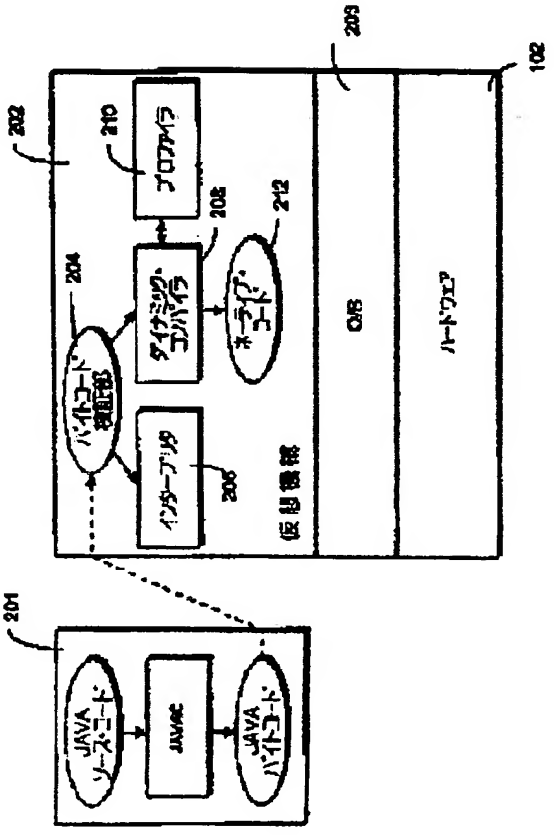
CODE GENERATION FOR BYTECODE COMPILER

Patent number: JP2000040007
Publication date: 2000-02-08
Inventor: SHAYLOR NICHOLAS
Applicant: SUN MICROSYST INC
Classification:
- international: G06F9/45; G06F9/44
- european:
Application number: JP19990184857 19990630
Priority number(s):

Also published as:
US6760907 (B2)
US2002104076 (A1)
GB2342473 (A)
DE19928980 (A1)
CA2274210 (A1)

Abstract of **JP2000040007**

PROBLEM TO BE SOLVED: To produce a native code in an execution time compiler from a bytecode group presented on a compiler and to provide a system which optimizes it.
SOLUTION: A compiler 208 accesses information 210 showing the possibility that a class is a specified type when an active program accesses. The compiler selects one code generation method among plural code generation methods by using the accessed information. A code generator produces an optimized native code in accordance with the selected code generation method and stores the optimized native code in a code cache 212 for reuse.



(19)日本国特許庁（ＪＰ）　(12) **公開特許公報（A）**　(11)特許出願公開番号
特開2000－40007
（P2000－40007A）
(43)公開日　平成12年２月８日(2000.2.8)

(51)Int.Cl.⁷	識別記号	FI		テーマコード（参考）
G06F 9/45		G06F 9/44	322F	
9/44	530		530P	

審査請求　未請求　請求項の数20　ＯＬ　（全 13 頁）

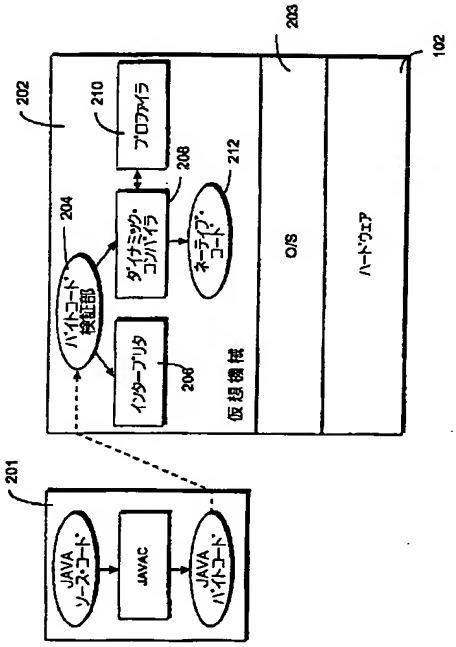
(21)出願番号　特願平11－184857
(22)出願日　平成11年６月30日(1999.6.30)
(31)優先権主張番号　09／108061
(32)優先日　平成10年６月30日(1998.6.30)
(33)優先権主張国　米国（US）

(71)出願人　597004720
サン・マイクロシステムズ・インコーポレーテッド
Sun Microsystems, Inc.
アメリカ合衆国カリフォルニア州94303, パロ・アルト, サン・アントニオ・ロード 901, エムエス・ピーエイエル01－521
(72)発明者　ニコラス・シェイラー
アメリカ合衆国カリフォルニア州94560, ニューアーク, ポトレロ・ドライブ 6167
(74)代理人　100089705
弁理士　社本　一夫　（外４名）

最終頁に続く

(54)【発明の名称】　バイトコード・コンパイラのためのコード生成

(57)【要約】
【課題】　コンパイラに提示したバイトコード群から、実行時コンパイラにおいてネーティブ・コードを生成し最適化するシステムを提供する。
【解決手段】　コンパイラ２０８は、実行中のプログラムがアクセスしたときに、クラスが特定のタイプである可能性を示す情報２１０にアクセスする。アクセスした情報を用いて、コンパイラは、複数のコード生成メソッドから、１つのコード生成メソッドを選択する。コード生成器は、選択したコード生成メソッドに応じて、最適化したネーティブ・コードを生成し、再利用のために、最適化したネーティブ・コードをコード・キャッシュ２１２に格納する。



【特許請求の範囲】

【請求項１】　バイトコードとして表現したプログラムにコードを生成するプロセスであって、
実行すべきメソッドを表す１群のバイトコードを受け取るステップと、
受け取ったバイトコードが、他のメソッドをコールするコール側メソッドを表す場合、ターゲット・メソッドのクラスが明白に認められるか否かについて判定を行うステップと、
前記ターゲット・メソッドのクラスが明白に認められる場合、前記ターゲット・メソッドからのコードを、実行すべきコール側メソッドのコードにインラインするコードを、前記受け取ったバイトコードのために生成するステップと、
前記コール側メソッドの後の実行のために、前記生成したコードをセーブするステップと、から成るプロセス。

【請求項２】　請求項１記載のプロセスにおいて、前記ターゲット・メソッドのクラスが曖昧である場合、前記メソッドが、更に、
前記ターゲット・メソッドのクラスが、サブクラス化することが認められるものか否かについて判定を行うステップと、
前記ターゲット・メソッドのクラスが、サブクラス化することが認められる場合、前記メソッドに対する仮想関数コールを実施するコードを、前記受け取ったバイトコードのために生成するステップと、を含むこと、を特徴とするプロセス。

【請求項３】　請求項１記載のプロセスにおいて、前記ターゲット・メソッドのクラスが曖昧である場合、前記メソッドが、更に、
前記ターゲット・メソッドのクラスが、サブクラス化することが認められるものか否かについて判定を行うステップと、
前記ターゲット・メソッドのクラスが、サブクラス化することが認められる場合、前記ターゲット・メソッドのクラスに可能な全てのサブクラスが認められるか否かについて判定を行うステップと、
前記ターゲット・メソッドのクラスに可能な全てのサブクラスが認められる場合、前記メソッドは、更に、
可能な各サブクラス毎にコード断片を生成するステップであって、各コード断片が、関連するサブクラスが定義するように、前記ターゲット・メソッドからのコードをインラインするステップと、
各コード断片毎にケースを有するスイッチを実施するコードを、前記受け取ったバイトコードのために生成するステップと、
前記コール側メソッドを実行した時点における前記ターゲット・メソッドの現サブクラス状態に基づいて、前記スイッチを選択するコードを、前記受け取ったバイトコードに生成するステップと、を含むこと、を特徴とするプロセス。

【請求項４】　請求項１記載のプロセスにおいて、前記ターゲット・メソッドのクラスが曖昧である場合、前記メソッドが、更に、
前記ターゲット・メソッドのクラスとして正しい可能性があるクラスの集合を識別するステップと、
前記コール側クラスのあらゆる実行について、選択したクラスが正しい可能性が、正しくない可能性よりも高いという知識に基づいて、前記可能なクラス集合から１つのクラスを選択するステップと、
前記選択したクラスからのコードを、実行すべき前記コール側メソッドのコードにインラインする、第１コード断片を生成するステップと、
コードをインラインせずに、仮想関数コールを実施する第２のコード断片を生成するステップと、
前記第１および第２のコード断片の前に、前記コール側メソッドの各実行毎に、選択したクラスが正しいクラスか否かについて検査するように動作する検査コードを生成するステップと、
前記検査コードに応答して、前記第１のコード断片または前記第２のコード断片にいずれかを、コード実行に選択させる分岐コードを生成するステップと、
前記第１コード断片、第２コード断片、検査コードおよび分岐コードを結合し、前記受け取ったバイトコードのために生成した前記コードを形成するステップと、を含むこと、を特徴とするプロセス。

【請求項５】　請求項１記載のプロセスにおいて、前記ターゲット・メソッドのクラスが曖昧である場合、前記メソッドが、更に、
前記ターゲット・メソッドのクラスを固定化したかのように、前記ターゲット・メソッドのクラスからコードをインラインする第１のコード断片を、前記受け取ったバイトコードのために生成するステップと、
前記ターゲット命令に対する仮想関数コールを実施し、プログラムの実行を前記第１のコード断片に戻す、第２のコード断片を生成するステップと、
前記ターゲット・メソッドに対するエントリを備えたデータ構造を作成し、前記エントリに、前記第１のコード断片における第１の命令の第１のアドレスと、前記第２のコード断片の第１の命令の第２のアドレスを格納するステップと、
前記ターゲット・メソッドを無効化する毎に、前記第１のコード断片の前記第１の命令に、プログラムの実行を前記第２のコード断片に転送する第３のコード断片をパッチするステップと、を含むこと、を特徴とするプロセス。

【請求項６】　請求項５記載のプロセスにおいて、前記第３のコード断片が分岐命令を備え、前記第２のコード断片が、プログラムの実行を前記第１コード断片の終端に戻す分岐命令を含むこと、を特徴とするプロセス。

【請求項７】　請求項５記載のプロセスにおいて、前記第３のコード断片がコール命令を備え、前記パッチングするステップが、更に、前記第１のコード断片における第２の命令を、プログラムの実行を前記第１のコード断片の終端に戻す分岐命令と置換するステップを含むこと、を特徴とするプロセス。

【請求項８】　請求項５記載のプロセスにおいて、前記第３のコード断片が、ブレークポイント・ハンドラ・ルーチンを呼び出す、ブレークポイント命令を備えること、を特徴とするプロセス。

【請求項９】　請求項８記載のプロセスにおいて、前記ブレークポイント・ハンドラ・ルーチンが、プログラムの実行を前記第１のコード・フラグメントの終端に戻すコードを含むこと、を特徴とするプロセス。

【請求項１０】　請求項１記載のプロセスにおいて、前記ターゲット・メソッドのクラスがサブクラス化することが認められるものか否かについて判定を行う前記ステップが、実行すべき前記命令の以前の実行の履歴を試験するステップを含むこと、を特徴とするプロセス。

【請求項１１】　受け取ったバイトコードからコードを生成する装置であって、

前記バイトコードを受け取るコンパイラと、

前記コンパイラ内にあり、複数のエントリを有するデータ構造であって、各エントリが、前記バイトコードの一意の群によって参照されるオブジェクト指向プログラム・クラスに対応する、データ構造と、

前記データ構造の各エントリ内にあり、前記対応するクラスが、サブクラス化する可能性が高いか否かを示すデータ・レコードと、

前記コンパイラ内にあり、前記データ・レコードにアクセスするように結合したコード生成器であって、各バイトコード群毎に、当該バイトコード群が参照するクラスに対応する前記データ・レコードの現在値に基づいて、複数のコード生成オプションの中から選択するコード生成器と、

複数のエントリを有するネーティブ・コード・キャッシュであって、各エントリが特定のバイトコード群に対応し、各エントリが、前記特定のバイトコード群と関連付けて生成したネーティブ・コードを保持し、前記コンパイラが特定のバイトコード群を最初に受け取る毎に、前記バイトコード群を前記コード生成器に渡し、前記コンパイラが前記特定のバイトコード群を後に受け取る毎に、前記ネーティブ・コード・キャッシュから選択したコードに、前記バイトコードを変換する、ネーティブ・コード・キャッシュと、から成る装置。

【請求項１２】　請求項１１記載の装置であって、更に、

ある時間期間にわたって前記コンパイラを監視するように結合してあり、プログラムの実行の間に呼び出された実際のクラスおよびサブクラスに関する知識を蓄積するプロファイラを備えること、を特徴とする装置。

【請求項１３】　オブジェクト指向プログラム・コードの実行時コンパイル・システムであって、前記プログラム・コードが、複数のオブジェクト指向クラスを定義するコードを含み、

前記プログラム・コードを受け取り、ターゲット・メソッドをコールするコール側メソッドを識別し、前記ターゲット・メソッドのクラスが明確に認められるか否かについて判定を行う実行時コンパイラと、

前記コンパイラ内にあり、前記ターゲット・メソッドからのコードを、実行すべきコール側メソッドのコードにインラインするコード生成器と、

前記コール側メソッドのその後の実行のために、生成したコードを格納するデータ構造を備えたコード・キャッシュと、を備えること、を特徴とするシステム。

【請求項１４】　コンピュータ・システムであって、

バイトコード・プログラムを格納するメモリであって、該バイトコード・プログラムが一連のバイトコードを含み、複数のバイトコード群が、コール側メソッドおよびターゲット・メソッドを含む、複数の実行すべきメソッドを表し、前記コール側メソッドが、前記ターゲット・メソッドをコールする動作を含む、メモリと、

ネーティブ・コードを実行するデータ処理ユニットと、

前記メソッドの少なくともいくつかのネーティブ・コード表現を格納するコード・キャッシュと、

特定のバイトコード群を最初に受け取ったときに、当該受け取ったバイトコード群を、前記バイトコードのネーティブ・コード表現に変換し、該ネーティブ・コード表現を前記コード・キャッシュに格納するように動作するコンパイラと、

前記データ処理ユニットに結合したプロファイラであって、各メソッドのクラスについて、当該メソッドのクラスがサブクラス化する可能性を示す情報を格納するデータ構造を有する、プロファイラと、

前記コンパイラ内にあり、前記プロファイラのデータ構造に結合してあり、前記格納した情報に基づいて、コード生成メソッドを選択する、最適化コードと、から成るコンピュータ・システム。

【請求項１５】　請求項１４記載のコンピュータ・システムにおいて、前記プロファイラが、更に、各メソッドの実行を監視し、各メソッドのクラスがサブクラス化するときを記録するモニタを備え、

前記コンピュータ・システムが、更に、前記プロファイラのデータ構造内に格納した情報を変更し、前記記録したモニタ情報を示す履歴アキュムレータを備えること、を特徴とするコンピュータ・システム。

【請求項１６】　請求項１４記載のコンピュータ・システムにおいて、前記最適化コードが、更に、

プロファイラのデータ構造に結合してあり、メソッドのクラスが明確に認められる場合を識別するコードと、

プロファイラのデータ構造に結合してあり、メソッドのクラスが動的なタイプであるが、特定のタイプである可能性の方がそうでない可能性よりも高い場合を識別するコードと、
プロファイラのデータ構造に結合してあり、メソッドのクラスが動的であるが、いずれの記録した実行においてもサブクラス化されたことが全くない場合を示すコードと、
プロファイラのデータ構造に結合してあり、メソッドのクラスが動的であり、いずれの特定の実行時においても、複数のタイプのいずれか１つである可能性が高い場合を識別するコードと、を備えること、を特徴とするコンピュータ・システム。
【請求項１７】　請求項１６記載のコンピュータ・システムにおいて、前記コンパイラが、更に、
コードが明確に認められるという識別に応答し、いずれかのコール側メソッドのためにコードを生成する場合に、前記メソッドを表すネーティブ・コードを、前記コール側メソッドのネーティブ・コードにインラインするコンパイル方法を備えること、を特徴とするコンピュータ・システム。
【請求項１８】　請求項１６記載のコンピュータ・システムにおいて、前記コンパイラが、更に、メソッドのクラスが動的なタイプであるが、特定のタイプのものである可能性の方がそうでない可能性よりも高いという識別に応答し、
正しい可能性の方が正しくない可能性よりも高いと識別された前記特定のタイプの前記クラスから前記メソッドを識別するネーティブ・コードをインラインし、
前記メソッドのクラスの現タイプに対する仮想関数コールを実施し、
正しい可能性の方が正しくない可能性よりも高いと識別された前記特定のタイプの前記クラスが実際には正しいか否かを検査し、
前記検査に基づいて、前記インライン・ネーティブ・コードまたは前記仮想関数コール・コードのいずれかに分岐する、コードを生成するコンパイル方法を備えること、を特徴とするコンパイル・システム。
【請求項１９】　請求項１６記載のコンピュータ・システムにおいて、前記コンパイラが、更に、
メソッドのクラスが動的であるが、いずれの記録した実行においてもサブクラス化したことが全くないという識別に応答し、前記ターゲット・メソッドのクラスから前記ターゲット・メソッドを表すネーティブ・コードをインラインする第１のコード断片と、前記ターゲット・メソッドのクラスに対する仮想関数コールを実施する第２のコード断片とを生成するコンパイル方法と、
前記第１のコード断片を保持する、前記コード・キャッシュ内の第１の位置と、
前記第２のコード断片を保持する、前記コード・キャッシュ内の第２の位置と、
前記第１の位置のアドレスと前記第２の位置のアドレスとを保持するデータ構造と、
前記ターゲット・メソッドのクラスがサブクラス化したか否かについて検出するコードと、
前記ターゲット・メソッドのクラスがサブクラス化したことの検出に応答し、前記コード・キャッシュ内の前記第１の位置のアドレスに、ブレークポイント命令をパッチするコードと、
前記コード・キャッシュ内の前記第２の位置にあるコードを実行し、前記コード・キャッシュ内の前記第１の位置の終端のアドレスに戻る、ブレークポイント処理ルーチンと、を備えること、を特徴とするコンピュータ・システム。
【請求項２０】　コンピュータに結合する搬送波に具体化し、バイトコードとして表すプログラムのために、前記コンピュータにおいてコードを生成するコンピュータ・データ信号であって、
前記コンピュータに、実行すべき方法を表すバイトコード群を受け取らせるように構成したコードから成る第１のコード部分と、
前記コンピュータに、前記ターゲット・メソッドの各実行毎に、前記ターゲット・メソッドのクラスが特定のタイプのものである可能性が高いか否かについて判定を行わせるように構成したコードから成る第２のコード部分と、
前記第２のコード部分に結合してあり、前記コンピュータに、前記第２のコード部分が行った決定に基づいて、コード生成計画を選択させるように構成したコードから成る第３のコード部分と、
前記第３のコード部分に結合してあり、前記選択したコード生成計画に応じて、前記バイトコード群を実装するコードを生成する第４のコード部分と、を備えるコンピュータ・データ信号。
【発明の詳細な説明】
【０００１】
【発明の属する技術分野】本発明は、一般的に、コンパイラに関し、更に特定すれば、バイトコードとして表現したプログラムの実行を最適化するためのコード生成技法に関するものである。
【０００２】
【従来の技術】Ｊａｖａ™プログラミング言語（Sun Microsystems Inc.（サン・マイクロシステムズ社の商標）のようなバイトコード・プログラミング言語は、バイトコードの集合として、コンピュータ・プログラムを表現する。各バイトコードは、クライアント・コンピュータ上のソフトウエアにのみ存在する「仮想機械」のための数値機械コードである。仮想機械とは、本質的にはインタープリタであり、バイトコードを理解し、このバイトコードを機械コードに翻訳し、次いでネーティブ機

械（native machine）上でこの機械コードを実行する。
【０００３】Ｊａｖａプログラミング言語のようなバイトコード・プログラミング言語は、ソフトウエア・アプリケーション開発者の間で好評を勝ち得ている。何故なら、これらは、使用が容易でありしかも移植性が高いからである。バイトコードとして表現したプログラムは、バイトコードを適正に解釈し変換可能な仮想機械を有するコンピュータであれば、いずれにも容易に移植することができる。しかしながら、バイトコード・プログラミング言語は、実行時にクライアント・コンピュータ上で解釈しなければならないので、ＣまたはＣ＋＋のような従来からのコンパイル型言語と勝負になる速度の実行は不可能であるという欠点があった。
【０００４】バイトコード言語の速度の限界は、主に、コンパイル・プロセスに関係する。コンパイルは、高水準言語（即ち、人が読むことができる言語）で著したプログラムを機械読み取り可能コードに変換するプロセスのことである。コンパイル・プロセスには、４つの基本的な工程がある。即ち、トークン化（tokenizing）、解析、コード生成、および最適化である。従来のコンパイル型プログラムでは、これらの工程は全て、実行時前に完了する。これに対して、ＢＡＳＩＣのようなインタープリタ型言語では、コンパイル工程の全ては、実行時に各命令毎に行う。ＣＳＨのようなコマンド・シェル（command shell）も有限数のコマンドを認識するインタープリタの例である。インタープリタ型言語は、得られたコードを最適化する方法がないので、非効率となる。
【０００５】バイトコード・プログラミング言語では、トークン化および解析は、実行時前に行われる。解析後に、仮想機械が解釈できるバイトコードにプログラムを変換する。その結果、バイトコード・インタープリタは、元来のＢＡＳＩＣプログラミング言語の実施のいくつかにおけるような言語インタープリタよりは高速となる。また、得られたプログラムは、バイトコード・フォーマットで表現した場合、完全にコンパイルしたプログラムよりも簡潔である。これらの特徴のために、バイトコード言語は、実行のためにソフトウエアをある機械から別の機械に転送することがあるネットワーク状コンピュータ環境では、有用な折衷案となった。
【０００６】Ｊａｖａプログラミング環境では、バイトコードへの変換を行うプログラムのことを「ｊａｖａｃ」と呼ぶが、Ｊａｖａコンパイラと呼ぶ場合もある（しかし、これはコンパイル・プロセスの一部を行うに過ぎない）。クライアント・コンピュータ上でバイトコードを解釈するプログラムのことを、Ｊａｖａ仮想機械（ＪＶＭ）と呼ぶ。他のインタープリタと同様、ＪＶＭは、それが受け取った各バイトコードを実行するループ内で走る。しかしながら、バイトコードを解釈するので、仮想機械上でもなお時間のかかる変換工程がある。各バイトコード毎に、インタープリタは、対応する一連の機械命令を識別し、次いでこれらを実行する。いずれの変換に伴うオーバヘッドも、１回では取るに足らないが、実行する命令毎にオーバヘッドが蓄積する。大きなプログラムでは、オーバーヘッドは、完全にコンパイルした一連の命令を単純に実行する場合と比較すると、大量となる。その結果、Ｊａｖａプログラミング言語で書いた大きなアプリケーションは、同等のアプリケーションを完全にコンパイルした形態よりは遅くなる傾向がある。
【０００７】実行を高速化するために、仮想機械をジャスト・イン・タイム・コンパイラ即ちＪＩＴ（just-in-time）と結合したり、あるいはそれを内蔵している。ＪＩＴは、バイトコードを実行する前に、これらをネーティブ機械コードにコンパイルすることによって、バイトコード・インタープリタの実行時パフォーマンスを向上させる。ＪＩＴは、最初に一連のバイトコードを検出したときに、これらを機械命令に変換し、次いで機械命令を実行するので、バイトコードを順次呼び出して解釈することはない。機械命令は、メモリ内を除くと、どこにもセーブされないので、次回プログラムを走らせるとき、ＪＩＴコンパイル・プロセスを新たに開始する。
【０００８】その結果、バイトコードはなおも移植可能であり、しかも多くの場合、通常のインタープリタにおけるよりも、はるかに高速に走るようになる。ジャスト・イン・タイム・コンパイルは、特に、多くの計算プログラムにように、コード・セグメントを繰り返し実行する場合に有用である。また、ジャスト・イン・タイム・コンパイルは、パフォーマンスの改善には殆ど寄与せず、実際には、１回または数回実行した後再使用しない小さなコード部分では、パフォーマンスが遅くなる場合もある。
【０００９】ＪＩＴ技術の欠点の１つとして、コンパイルを実行時に行うので、機械コードを最適化しようとして費やす計算時間がオーバーヘッドとなり、プログラム実行の低速化を招く場合があることがあげられる。したがって、多くの最適化技法は、従来のＪＩＴコンパイラにおいては実用的でない。また、ＪＩＴコンパイラは、一度に大量のコードを見ないので、大量のコードを最適化することができない。この結果、コンパイラは、プログラムが用いたクラス集合の確実な判断ができない。更に、クラス集合は所与のプログラムを実行する毎に変化する可能性があるので、仮想機械コンパイラは、クラス集合が常に明確に認められると想定することが全くできない。
【００１０】この不確実性のため、実行時に真の最適ネーティブ・コードを生成することが困難となる。クラス集合の知識が不完全のまま最適化を試しても、非効率的となる可能性があり、あるいはプログラムの機能性を変えてしまう虞れもある。クラス集合に関する不確実性は、Ｊａｖａ実行の広い範囲にわたって潜在的な非効率

性を生ずる。クラス集合とは、Ｊａｖａ言語プログラムの特定のインスタンスを実行するために用いる全てのクラスの集合のことである。従来からのバッチ・コンパイル型プログラムでは、コンパイル時にプログラムが用いるクラス全体がわかるので、最適化の作業は非常に簡単である。

【００１１】共通プログラム発生（common program occurrence）とは、「メソッド・コール」と呼ぶプロセスにおいて、第１のメソッド（コール元メソッド）が第２のメソッド（ターゲット・メソッド）をコールすることである。このメソッド・コール・プロセスは、プログラマの観点からは利点があるが、多くのクロック・サイクルを消費する。オブジェクト指向プログラム・コードのコンパイルにおいて重要な最適化に、「インライニング」（inlining）と呼ぶものがある。完全にコンパイルしたプログラムでは、インライニングは、ターゲット・メソッドのコードをコール元メソッドにコピーすることによって、これらのターゲット・メソッドのコールを一層効率的に行う。

【００１２】しかしながら、Ｊａｖａプラットフォームの意味論（semantics）のため、コンパイラはクラス集合全体を全く判断することができない。クラスは拡張可能であり、Ｊａｖａに動的にロード可能であるので、後から発生するクラスの拡張が、メソッドを無効化してしまう可能性があり、コンパイラは、仮想メソッド・コールがいずれかの特定のメソッドに到達することを、確信を持って認めることができない。Ｊａｖａプログラミング言語では、デフォルトのメソッドは、「最終」と明示的に定義していなければ、無効化される可能性がある。「非最終」メソッドは全て、無効化可能と見なされる。

【００１３】したがって、非最終リーフ・メソッドに対する全てのコールは、ターゲット・メソッドが後のある時点で無効化される可能性があることを想定しなければならない。このため、以前はこれらを呼び出すために、時間のかかる仮想メソッド・コール・シーケンスを用いてきた。メソッドは小さい程、インライニングは一層有利となる。典型的に、１ライン・メソッドは、その内容を実行するよりも、ルーチンの開始および終了にはるかに時間がかかる。メソッドにオーバーロードするクラスが今後ないことを確実に知ることができれば、多くの場合これら非最終リーフ・メソッド・コールの８５％までもが解消可能であることが、検査によって明らかとなった。

【００１４】

【発明が解決しようとする課題】プログラムに対するクラス集合が明確に認められない環境においても、できる限り最適なネーティブ・コードを生成する方法および装置が必要とされている。また、クラス集合の知識が不完全な場合に対処し、プログラムの実行パフォーマンスへの影響を許容できる程度にする方法および装置も必要とされている。

【００１５】

【課題を解決するための手段】端的に述べると、本発明は、実行時コンパイラにおいて、当該コンパイラに提示する１群のバイトコードからネーティブ・コードを生成し最適化する方法、システムおよび装置に関する。コンパイラは、実行中のプログラムがクラスにアクセスするときに、そのクラスが特定のタイプである尤度を示す情報にアクセスする。このアクセスした情報を用いて、コンパイラは複数のコード生成メソッドから１つのコード生成メソッドを選択する。コード生成器は、選択したコード生成メソッドに応じて、最適化したネーティブ・コードを生成し、再利用のために最適化ネーティブ・コードをコード・キャッシュに格納する。

【００１６】

【発明の実施の形態】本発明は、同じバイトコード言語プログラムを繰り返し走らせるために用いるシステムが、「プロファイリング」（profiling）と呼ぶプロセスによって、これらのプログラムにおいて用いたクラス集合について、適切かつ正確な知識を蓄えることができるという前提を基にしている。プロファイリングは、単一のプログラムの実行、または異なるセッションで行われる多数のプログラムの実行に対して行うことができる。従来システムの欠点、即ち、実行時における特定の知識の欠如を利点に変えるのは、実行時に得られるこの知識の蓄積である。この利点が得られるのは、蓄積した知識が特定の実行時環境に特化され、実行中のプログラムのプロファイリングを行うときに実行時環境のあらゆる特質（idiosyncrasies）を本質的に考慮するからである。対照的に、従来のコンパイル・プロセスは、典型的な実行時環境における典型的なプログラムの挙動に対する一般化された知識に基づいて最適化を行っていた。

【００１７】本発明によるシステムは、以前の実行において用いられたクラスを拡張することはないことを「認めた」とき、または理由があってそう信じることができるときに、バイトコード・プログラムから効率的なネーティブ・コードを生成する。「認める（know）」および「知識（knowledge）」という用語は、ここで用いる場合、システムおよび／またはソフトウエア内のデータ構造が、既知の情報を表すデータを保持していることを意味する。同様に、理由がある確信とは、システムおよび／またはソフトウエアが、ある状態を確実に認めることができなくても、その状態は存在しないよりも存在する確率の方が高いことを示す情報を保持していることを示す。

【００１８】本発明は、実行中のプログラムについて認めた情報または理由があって確信した情報に応答し、ダイナミック・コンパイラを用いて、実行時にコードを生成する。特定の例では、ダイナミック・コンパイラは、多数のコード生成メソッドを含み、コンパイルしている

いずれの特定のバイトコード群についても、その中から選択することができる。どのメソッドをそれぞれの場合に用いるかという選択は、コード自体を検査することによって瞬時的に得ることができる知識、およびコードの実行を監視し経時的に収集した知識に基づいて行う。

【００１９】本発明は、プログラムにおける全てのバイトコードをネーティブ・コードにコンパイルするシステムに関して説明するが、本発明は、選択したバイトコード群のみをコンパイルするシステムに採用しても有用であることは理解されよう。例えば、本発明による仮想機械は、ダイナミック・コンパイラと並行に動作する、従来のバイトコード・インタープリタを含んでもよい。このような実現例では、選択したバイトコード群の実行頻度、これらのバイトコードをコンパイルすることによって達成し得る相対的な効果、またはその他のパフォーマンスに基づく評価基準を基に、これらをコンパイルすることができる。したがって、これらの実現例は、本発明の目的のためにここに記載する具体的な実現例と同等である。

【００２０】具体的には、本発明は、コードがターゲット・メソッド・コールを伴う場合に、インライニング技法を駆使することによってネーティブ・コードの最適化を図ることを目的とする。したがって、本発明は、オブジェクト指向プログラムの最適化において、他のオブジェクトからメソッドやそれらのメソッドを記述するクラスを頻繁にコールする、多くの小さなオブジェクトやメソッドを利用する場合に、特に有用である。Ｊａｖａプログラミング言語は、目的が限られた小さなメソッドを用いて厳格なオブジェクト定義を促進（encourage）する高度なオブジェクト指向言語の一例である。

【００２１】高度なオブジェクト指向言語では、クラス拡張性が、プログラミングを容易にし、機能性を改善する重要な特性（key feature）となる。クラスの拡張とは、これがサブクラス化され、このクラスが記述したメソッドが無効化されるときのことである。一旦無効化されると、無効になったメソッドに対する後続のコールは全て、新しいメソッドを用いるが、このメソッドは、メソッド・コールの受け取り側が拡張クラスである場合、親クラス即ち抽象クラス（abstract class）に元々記述されているメソッドではない。本発明の重要な有用性は、実行中のシステムにおいてサブクラス化を行うことに起因する複雑性に対処するシステムおよび方法を提供することにある。

【００２２】図1は、本発明による方法および装置を実施するように構成したコンピュータ・システム１００を示す。クライアント・コンピュータ１０２は、サーバ・コンピュータ１０４上に位置するプログラムをダウンロードする。クライアント・コンピュータは、プログラム命令を実行するプロセッサ・ユニット１０６を有する。プロセッサ・ユニット１０６は、システム・バスを通じてユーザ・インターフェース１０８に結合してある。ユーザ・インターフェース１０８は、ユーザに情報を表示するために使用可能なデバイス（例えば、ＣＲＴまたはＬＣＤディスプレイ）、およびユーザからの情報を受け入れるデバイス（例えば、キーボード、マウス等）を含む。メモリ・ユニット１１０（例えば、ＲＡＭ、ＲＯＭ、ＰＲＯＭ等）は、プログラム実行のためのデータおよび命令を格納する。記憶装置１１２は、大容量記憶装置（例えば、ハード・ディスク、ＣＤＲＯＭ、ネットワーク・ドライブ等）から成る。モデム１１４は、システム・バスからのデータを、ネットワーク１０５を介した伝送に適したフォーマットに、あるいはその逆に変換する。また、モデム１１４は、ネットワーク・アダプタ、あるいは通信ネットワーク用のその他の純粋なデジタル・アダプタまたは混成アナログ－デジタル・アダプタと交換しても同等である。

【００２３】サーバ１０４は、典型的に、同様のコンポーネント群から成り、プロセッサ１１６、ユーザ・インターフェース１１８、およびサーバ・メモリ１２０を含む。特定の例では、サーバ記憶装置１２２がバイトコードで表現したプログラムを格納し、このプログラムをモデム１１４を介しネットワーク１０５を通じてクライアント・コンピュータ１００に送信する。尚、本発明は、図1に示すネットワーク・コンピュータ環境以外にも、単一のコンピュータ上でも実施可能であることは理解されよう。その場合、バイトコード・プログラムを単にクライアント記憶装置１１２に格納すればよい。

【００２４】本発明の説明において、「クラス・タイプ」という用語は、特定のクラスを識別し、他のクラスまたは当該クラス自体のいずれのサブクラスとも、当該クラスを区別する特性のことを言う。クラスをサブクラス化した場合、新たなクラス・タイプが生ずる。特定のオブジェクトのクラス・タイプは、クラス記述に格納したデータから判定することができる。クラスは、「最終」キーワードを用いて定義することができるが、その場合クラスをサブクラス化することはできない。その他の場合、サブクラス化は可能である。オブジェクト指向プログラミングはサブクラス化を促進するので、大抵の場合（実際には、Ｊａｖａプログラミング言語におけるデフォルトの場合）、クラスはサブクラス化可能である。クラスがサブクラス化可能である場合、何回でもサブクラス化することができ、その結果、プログラムの実行全体に対して、クラス・タイプの集合が得られる。実行時に、メソッド・コールによってあるメソッドをコールする場合、このタイプ集合の内正しいのは特定のタイプ1つのみである。

【００２５】図2は、Ｊａｖａプログラミング環境の一例を示し、「コンパイル時環境」２０１、および「実行時環境」２０２を含む。Ｊａｖａ言語プログラムの開発および実行は、2つの工程を必要とする。プログラマ

は、Ｊａｖａソース・コードをタイプ入力する。これを、図２に示すｊａｖａｃのようなＪａｖａコンパイラによって、Ｊａｖａバイトコードと呼ぶ形態に変換する。前述のように、Ｊａｖａバイトコードは、簡潔でありしかも移植可能であるので、ネットワーク・コンピュータ・システムにおいてプログラムを格納したり転送するには、理想的な形態である。

【００２６】次に、バイトコードを実行時環境２０２に転送し、Ｊａｖａ仮想機械（ＪＶＭ）のような公知のプログラムによって実行する。全てのＪＶＭは同じバイトコードを理解するので、Ｊａｖａプログラムのバイトコード形態は、ＪＶＭを備えたあらゆるプラットフォーム上で走らせることができる。このように、ＪＶＭはＪａｖａバイトコードのための汎用実行エンジンである。所与のプラットフォームのためにＪＶＭを一旦書くことができれば、あらゆるバイトコード・プログラムをそれによって走らせることができる。従来のＪＶＭにおけると同様、好適な実現例は、バイトコード・インタープリタ２０６を含み、ダイナミック・コンパイラ２０８と並列に走って、コンパイラ２０８が時間を費やして最適化したネーティブ・コードを生成している間に、従来と同様に、解釈したネーティブ・コードを与える。

【００２７】本発明によるダイナミック・コンパイラ２０８は、前述のようにコンパイルを最適化することにより、Ｊａｖａコードの実行を高速化する。コンパイラ２０８は、Ｊａｖａバイトコードを取り込み、プロファイラ２１０から得た知識およびパフォーマンス情報に応答して、ユーザのハードウエア１０２の具体的な種別およびオペレーティング・システム２０３のためのネーティブ・コードに、バイトコードを変換する。本発明は、コンパイラ技術の最適化を実施し、プログラムをネーティブに実行可能な形態に変換することによって、高いパフォーマンスを得る。インタープリタ２０６からの解釈したコードとは異なり、コンパイラ２０８からの最適化コードは、後に再利用するために、コード・キャッシュ２１２にセーブしておく。

【００２８】プロファイラ２１０は、経時的にコードの実行を監視し、主に、実行中の特定のプログラムのサブクラス化およびオーバロード挙動を監視する。本発明の重要性として、プロファイラ２１０は、プログラムが実際にどのクラスを使用しているかに関する知識を蓄積する。プロファイラ２１０は、プロファイル・データを生成し、クラス毎またはメソッド毎に、プロファイル・データ構造（図示せず）に格納する。したがって、ダイナミック・コンパイラ２０８はクラス集合について先験的知識を全く有さないが、クラス集合およびクラス挙動に関してセッションに特化した知識を、プロファイラ２１０から得ることができる。実現例の一例では、プロファイラ２１０は、実行を監視する各クラス毎にエントリを有する、テーブルのようなデータ構造を含む。

【００２９】実行が進むに連れて、プロファイラ２１０は、データ構造にプロファイル・データを格納し、特定のクラスがサブクラス化されたか否かについて示すと共に、特定のクラスの１つのクラス・タイプが、いずれの特定メソッドの呼び出し時においても、正しいクラス・タイプである可能性が最も高いか否かについて示す。このように、格納したデータは、クラスの記述から直接判定可能な単純な「固定」指示よりも有用である。更に、プロファイル２１０内のデータ構造は、起動時に、何らかの既知状態に初期化することができるが、動的に変化して、これを用いている環境におけるプログラムの現挙動を反映している。

【００３０】本発明は、メソッド・コールの内、本発明が特に有用な４つのカテゴリに関して理解すると役に立つ。第１のケースは、受け取り側のオブジェクトのクラス（即ち、ターゲット・メソッドを定義するクラス）が明確に認められる場合のメソッド・コールである。これは、メソッドが「最終」として示されたとき（即ち、その記述の最終というキーワードを含む場合）、または、明示的に最終と宣言されたのではないが、実際には当該メソッドが最終であることが固定コード分析によって解明されたときに、得ることができる。第２のケースでは、受信側のクラスが、可変であることが認められている。抽象クラスに定義してあるメソッド、またはサブクラス化することが認められているクラスへのコールが、このケースの例である。これら最初の２つのケースは、クラスが明確に固定であることが認められる場合、および特定の実行においてどのタイプとなるかについてクラスが非常に曖昧である場合という、極端なケースを表す。

【００３１】残りの２つのケースは、クラスに関する何らかの実行時知識を利用して、より効率的なコードを生成可能な、不明確な領域（gray area）を含む。第３のケースでは、受け取り側のクラスは、ある特定のタイプである場合の方がそうでない場合よりも多いものとして識別する。本発明によれば、このケースは、プロファイルに基づくフィードバックによって識別することができる。第４のケースでは、受け取り側のクラスは、以前の実行において全くサブクラス化したことがないことを、プロファイルに基づくフィードバックが示したことにより、このクラスは特定のタイプであることの可能性が非常に高い。

【００３２】本発明によれば、第１のケースでは、後続のメソッド・コールによって、クラスが無効化され得ないことがわかっている。この場合、コール側に対して生成したコードに、メソッド・コールを常に直接コールするか、またはインラインすることができる。これら２つのタイプの最適化（直接コールおよびインライニング）は、別個であるが、類似した動作である。直接コールは、仮想メソッド・コールよりも実行が速い。更に高速

化した最適化は、ターゲット・メソッドのコードをコール側メソッドのコードに実際にコピーすることである（即ち、インライニング）。しかしながら、インライニングは、実行時にコード・サイズの増大を招くため、必要なコンピュータの命令キャッシュが多くなる。特に指定しない限り、これら２つの最適化技法間の選択は、個々の用途の必要性を満たすのに適したコードの生成に係ることである。

【００３３】ケース２では、本発明を単純に実施すると、従来の仮想機能コールを実施するコードを生成することになる。最適化はこのような場合にはより多くのオーバヘッドを必要とし、その結果メソッドのコール毎に非常に大量のコードを生成する場合があるので、これはもっともなことである。あるいは、全ての可能なクラスを「スイッチ・ステートメント」（switch statement）における場合として識別し、各可能なクラスからのターゲット・メソッドのコードを、コール側メソッドに対するコードにインラインする。スイッチは、現クラス・タイプを検査し、検査結果に基づいてスイッチを選択することにより、コール元メソッドを実行するときに選択する。

【００３４】第３のケースでは、コールは、ターゲット・メソッドのコードにおいて小さな検査を行うことによって、最も正しい可能性が高いメソッドに直接行うことができる。クラスがそのメソッドに対する正しいクラスでない場合、仮想関数コールを行うことができる。あるいは、ケース３のメソッドは、インラインすることができるが、インラインするコードの最初の命令の前に検査を行い、クラス・タイプが正しいか否かチェックし、正しくない場合代わりに仮想メソッド・コールを行う。

【００３５】ケース４では、メソッドのコールは、メソッドを直接コールすることによって効率的に行うことができる。本発明によれば、生成するコードは、クラスが決してサブクラス化しないことを始めに想定する。しかしながら、後にこれがサブクラス化した場合、メソッドの最初の命令を小さなコード・フラグメントにパッチし（patch）、受信側を検査し、それが正しくない場合仮想関数コールを行う。それが正しい場合、パッチした命令を実行し、分岐して主メソッド・コードに戻る。

【００３６】ターゲット・メソッドが短い程、インラインすることは一層重要となる。コール側メソッドの再コンパイルを回避するためには、受け取り側クラス（即ち、ターゲット・メソッドのクラス）の検査が必要となる。望ましくは、この検査はオーバヘッドを殆ど追加せず、比較的素早いとよい。単一命令ターゲット・メソッドをコールするメソッドの最悪の場合の例を、単純なアクセッサ・メソッド（accessor method）によって例証する。例えば、以下の記述を有するクラスＹを想定する。

【００３７】

【表１】

class Y having the description:

```
class Y {
    int value;              //declare "value" as an
integer                                 //data type
    int length() {          //declare length() as a
method              return value;   //that returns
an integer "value"
      }                     //when called
}
```

【００３８】例えば、コール側メソッドは、

```
x=y.length();
```

とすることができる。

【００３９】このコール側メソッドのために生成したネーティブ・コード・セグメントは、次のようになる。

【００４０】

【表２】

```
cmp [y+class],#constantForClassY        //compare y's
class type

                                          //to a
specified constant
bne vfCall                              //branch when
the

                                          //comparison
is not

                                          //equal
mov[y+value},x                        //move the
returned

                                          //"value" to x
```

【００４１】上述機械コードでは、最初の命令は、メモリ内のクラス記述から読み出したクラス・タイプを、予期するクラスに対する所定の定数と比較する。この所定の定数は、コードを生成するときに決定し、コードを再生成するときはいつでも更新することができる。２つの値が等しい場合、等しくないときに分岐する命令（branch-on-not-equal instructuion）を飛ばし、インライン・メソッドを実施する機械コードを実行する。仮想関数テーブルのコール・スルー（call-through）の場合は、ｖｆＣｃａｌｌにおいて処理し、その結果何らかの影響がパフォーマンスに現れる。これは、異常な場合と考えられるので、パフォーマンスには問題はない。

【００４２】このクラス・タイプ検査を多数のメソッド・コールに再利用可能とするように、前述のコードを編成し得ることも考えられる。同じクラスのいくつかのコール・メソッドおよびターゲット・メソッドのクラスが、あるコールから他のコールにサブクラス化できなかった場合、単一のクラス・タイプ検査を行い、この検査をターゲット・メソッド・コール全てで使用することができる。この実現例は、クラス・タイプ検査を実行することに伴うオーバヘッドを、複数のメソッド・コールに広げ、これによって効率の向上を図る。

【００４３】更にインライニングの改善を得る方法は、クラス検査（例えば、上述の例では、ＣＭＰおよびＢＮＥ命令）を全て除去することである。その結果得られるコードは、ターゲット・メソッドのコールのみを含み、したがって非常に効率的となる。しかしながら、ターゲット・メソッドが既に無効化されている場合、インラインされたコードは正確でない。したがって、これを行う場合、ターゲット・メソッドのクラスが、そのメソッドを無効化したクラスによってサブクラス化されている場合、そのインラインされたコード・セグメントを仮想コール・シーケンスで置換しなければならないことを、ＪＶＭは思い出さなければならない。

【００４４】仮想コール・シーケンスは、それと置換するいずれのものよりも大きい可能性があり、したがってこれを解決するためには、元のインライン・コードの最初の命令を、仮想コール・シーケンスを実施する命令がある位置にプログラムの実行を転送する命令に変更することができる。その例には、分岐命令、コール命令、ブレークポイント命令等が含まれる。どのようにしてこの特徴を実施するかについての選択は、個々の機械に適した効率的で柔軟性のある実施を与えるのは何かに基づいて行う。

【００４５】図３に示す分岐の実施では、インライン・コードの最初の命令を、分岐命令（例えば、ＢＲ　Ｘ。Ｘは仮想関数コールを実施するコードの位置である）と置換する。後続の実行時に、インライン・コードを実行せず、代わりに位置Ｘのコードを実行する。仮想関数コールを実施するコードは、インライン・コード・セグメントの終端に戻る分岐命令で終了しなければならない。

【００４６】図４に示す「コール」の実施では、インライン・コードの最初の命令は、仮想関数コールを実施する位置Ｘにおける命令をコールするコール命令と置換する。コールしたシーケンスの終了時に、リターン命令（図４におけるＲＴＮ）が、パッチしたコードにおける丁度コール命令の後ろの位置に、命令の実行を戻す。インライン・コードの２番目の命令には、インライン・コード・セグメントの終端を指し示す分岐命令をパッチする。

【００４７】図４に示すコールの実施は、コンピュータの仮想メモリ空間（例えば、ＳＰＡＲＣアーキテクチャの場合、４ギガバイトのアドレス空間）内のどこにでも、コールした命令シーケンスを置くことができるようにコール命令を実施する、ＳＰＡＲＣアーキテクチャ・プロセッサのような、いくつかのプロセッサでは非常に効率的で柔軟性がある。対照的に、典型的な分岐命令は、当該分岐命令周囲の限られたアドレス空間範囲内にある位置に分岐できるに過ぎない。これは、パッチ・コードを実際に実行時にコンパイルするような場合には、重要な利点となる。例えば、図３および図４の位置Ｘにあるコードを必要とすると判定した後（即ち、インライ

ン・コードの元になったメソッドが無効化された後）にようやくこれをコンパイルした場合、パッチング・コードを保持するための空メモリ空間が近くにない場合がある。したがって、パッチング・コードは、分岐命令を通じてアクセス可能なアドレス範囲の外側に位置付けなければならない場合がある。あるいは、プログラムには、必要に応じてパッチング・コードを保持するために使用可能な空のメモリ空間を満載することができるが、しかしながら、これはメモリ資源の使用法としてはむしろ非効率的であり、プログラムの大型化（bloat）や、命令キャッシュの管理の複雑化を招く虞れがある。

【００４８】図5は、オペレーティング・システムのブレークポイント・ハンドラを用いて、パッチング・コードを実施する実現例を示す。図5に示すように、インライン・コードの最初の命令に、実行時に例外を投入するブレークポイント命令をパッチする。ブレークポイント命令は、ブレークポイント・ハンドラが捕獲し、ブレークポイント・テーブル６０１のようなデータ構造を参照する。各ブレークポイント命令「Ａ」毎に、テーブル６０１はブレークポイントを処理するための１組の命令「Ｂ」をマップする。一方、Ｂにおける命令は仮想関数コールを実施することができる。尚、このプロシージャは非常に遅いシーケンスであるが、このケースが稀にしか起こらない用途環境では、正味のパフォーマンス改善が得られる。これらの非効率性は、プロファイルに記録しておき、次回タスクを走らせる際に補正することができる。

【００４９】この機構を可能にするために、コンパイラがメソッドを作成するとき、インライン・コード自体を実施するコード断片だけでなく、仮想関数コールを実施するコード断片も作成しなければならない。この仮想関数コール・コード断片は、ブレークポイント・ハンドラがコールし、セーブしてあるリターン・アドレスを増分し、インラインされたが現在では無効のコードの終点に、コードの実行をジャンプさせ、次いで割り込みシーケンスからのリターンを行わせなければならない。パッチするメモリ位置のアドレス、およびコード断片を保持するメモリ位置のアドレスは、インラインするメソッドと関連するデータ構造に置く。このメソッドがサブクラス化するときはいつでも、命令データ構造内の命令にパッチを行い、ブレークポイント・ハンドラに、その関連付けを通知する。

【００５０】本発明によるシステムおよび方法に関する魅力的な特徴は、コードを一旦生成したなら、再度コンパイルする必要が全くなく、難しいスタック・パッチング・プロシージャが不要であることである。コンパイラの複雑性およびデータ構造のサイズが多少増大するが、コード実行効率の改善は明白である。

【００５１】以上、ある程度の特定性をもって本発明を説明し例示したが、本開示は一例として行ったに過ぎず、特許請求の範囲における本発明の精神および範囲から逸脱することなく、当業者には、その部分の組み合わせや構成において数多くの変更を行うことが可能であることは理解されよう。

【図面の簡単な説明】

【図１】本発明による方法およびシステムを実施するネットワーク・コンピュータ環境を示す図である。

【図２】本発明のシステムおよび方法による、プログラミング環境を示す図である。

【図３】本発明の実現例の一例を示す図である。

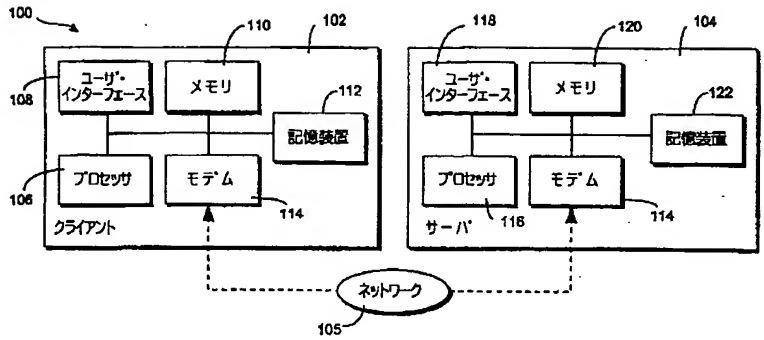
【図４】本発明による別の実施形態を示す図である。

【図５】本発明による更に別の実施形態を示す図である。

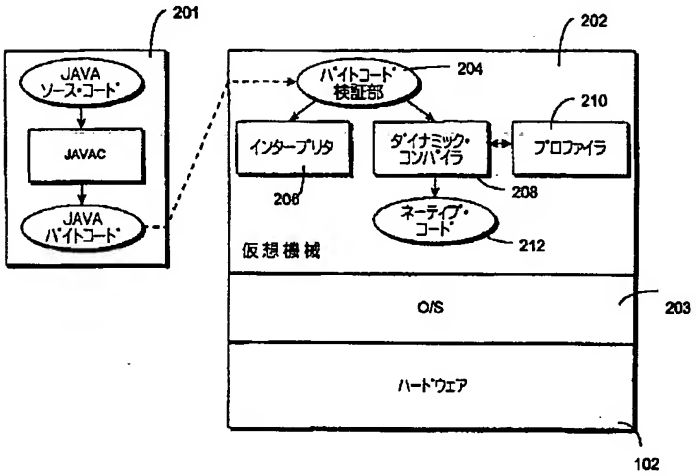
【符号の説明】

１００　コンピュータ・システム
１０２　クライアント・コンピュータ
１０４　サーバ・コンピュータ
１０５　ネットワーク
１０６　プロセッサ・ユニット
１０８　ユーザ・インターフェース
１１０　メモリ・ユニット
１１２　記憶装置
１１４　モデム
１１６　プロセッサ
１１８　ユーザ・インターフェース
１２０　サーバ・メモリ
１２２　サーバ記憶装置
２０１　コンパイル時環境
２０２　実行時環境
２０３　オペレーティング・システム
２０４　バイトコード検証部
２０６　バイトコード・インタープリタ
２０８　ダイナミック・コンパイラ
２１０　プロファイラ
２１２　コード・キャッシュ

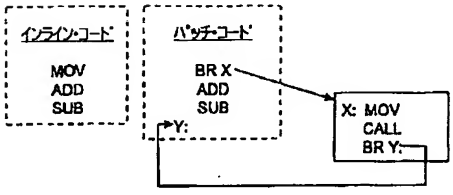
【図1】


100
110
102
118
120
104
ユーザ・インターフェース
メモリ
108
112
記憶装置
122
プロセッサ
モデム
106
114
116
クライアント
サーバ
ネットワーク
105

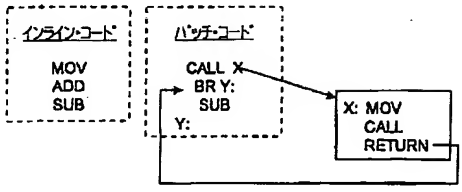
【図2】


201
202
JAVA ソース・コード
JAVAC
JAVA バイトコード
バイトコード検証部
204
210
インタープリタ
ダイナミック・コンパイラ
プロファイラ
206
208
ネーティブ・コード
212
仮想機械
O/S
203
ハードウェア
102

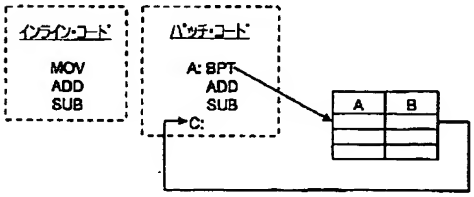
【図3】


インライン・コード
MOV
ADD
SUB
パッチ・コード
BR X
ADD
SUB
Y:
X: MOV
CALL
BR Y:

【図4】


インライン・コード
MOV
ADD
SUB
パッチ・コード
CALL X
BR Y:
SUB
Y:
X: MOV
CALL
RETURN

【図5】


インライン・コード
MOV
ADD
SUB
パッチ・コード
A: BPT
ADD
SUB
C:
A
B

フロントページの続き

(71)出願人 597004720
2550 Garcia Avenue, MS PAL1-521, Mountain View, California 94043-1100, United States of America